DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs*

Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho,
Milan Lenčo, Petr Ročkai**, Vladimír Štill, and Jiří Weiser

Faculty of Informatics, Masaryk University
Brno, Czech Republic
divine@fi.muni.cz

Abstract. We present a new release of the parallel and distributed LTL model checker DiVinE. The major improvement in this new release is an extension of the class of systems that may be verified with the model checker, while preserving the unique DiVinE feature, namely parallel and distributed-memory processing. Version 3.0 comes with support for direct model checking of (closed) multithreaded C/C++ programs, full untimed-LTL model checking of timed automata, and a general-purpose framework for interfacing with arbitrary system modelling tools.

1 Introduction

Even though explicit-state model checking is a core method of automated formal verification, there are still major roadblocks, preventing the software development industry from fully utilising explicit-state model checkers. One is the well-known state space explosion problem, which restricts the size of systems that can be efficiently handled by a model checker. Another, possibly even more serious, is the requirement to create a separate model of the system, disconnected from its source code. This adds a substantial amount of work to the process of model checking, increasing its price and making the method less feasible industrially. The problem is compounded by relative obscurity of modelling languages.

In version 3.0, DiVinE [2–5] addresses both these problems: based on a newly developed LLVM bitcode interpreter, it can directly verify closed C/C++ programs, eliminating the extra human effort directed at modelling the system. At the same time, DiVinE 3.0 offers efficient state-space reduction techniques (Partial Order Reduction, Path Compression), combined with parallel and distributed-memory processing. This makes DiVinE suitable for verification of large systems, especially when compared to more traditional, sequential model checkers.

[*] This work has been partially supported by the Czech Science Foundation grant No. GAP202/11/0312.

[**] Petr Ročkai has been partially supported by Red Hat, Inc. and is a holder of Brno PhD Talent financial aid provided by Brno City Municipality.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 863–868, 2013.

2 Engine Improvements Since DiVinE 2.5

While the primary focus of the 3.0 release was on language support, there have been important improvements in the model-checking core as well. A major addition is the optional use of hash compaction and disk-based queues, designed to work hand-in-hand to reduce memory footprint. While hash compaction introduces a small risk of missing counter-examples, and hence results obtained with hash compaction cannot guarantee correctness, it has proven to be extremely useful in tracking down bugs in large, complex systems that cannot be entirely verified at reasonable expense with available technology. As implemented in DiVinE, hash compaction can be used with both reachability analysis and LTL model checking and is compatible with distributed-memory verification. [6]

While algorithms using traditional static partitioning and per-thread hash tables provide reasonable scalability, a single shared hash-table and dynamic work partitioning can give substantially better results, as has been demonstrated by LTSmin [9]. Hence, DiVinE 3.0 provides an experimental mode of operation using a single shared hash table. While this mode is a proof of concept and is not recommended for production use in this release, future 3.x versions of DiVinE will integrate it more tightly.

3 DVE: The Native Modelling Language

The DVE language was conceived and implemented in the early phases of development of DiVinE. Since then, it became successful in its own right as a simple yet still powerful formalism for modelling asynchronous systems and protocols. Nevertheless, the original implementation has been falling out with rapid development in other parts of DiVinE. In version 3.0, we have replaced the legacy DVE interpreter with a modern, more flexible and extensible design. Gradual, backward-compatible improvements to the DVE language are expected in the 3.x line of development.

In addition to an improved interpreter, DiVinE 3.0 has added an ability to restrict LTL model checking to (weakly) fair runs. This feature is so far unique to the DVE language, although future extensions to other input languages are planned.

4 LLVM: Model Checking Multithreaded C++

The major highlight of the new version of DiVinE is the ability to directly model-check LLVM bitcode. This in turn enables programmers to use DiVinE for model checking of closed C and C++ programs, since major C and C++ compilers[1] can produce LLVM bitcode.

[1] Clang and GCC (with a plugin) can generate both optimised and unoptimised LLVM bitcode. Compilers for other languages are available as well.

Table 1. Efficiency of LLVM bitcode reductions

	state space reduction			
model, flags	none	τ	τ+	all
`peterson.c, -O0`	294193	2181	596	212
`peterson.c, -O1`	33227	491	286	278
`peterson.c, -O2`	21122	443	268	260

Userspace programs normally needs to be linked to system libraries for execution; while purely computational fragments of system libraries can be directly translated into LLVM bitcode and linked into the program for verification purposes, this is not the case with "IO" facilities (including any calls into the OS kernel). For some of these, DiVinE provides substitutes – most importantly the POSIX thread API, while other may need to be provided by the user, possibly implemented in terms of a nondeterministic choice operator (`__divine_choice`) provided by DiVinE. This means that no IO is possible (but it may be substituted by nondeterminism) and this automatically makes the program closed. Hence, no other "special" treatment is required to verify programs.

Since DiVinE provides an implementation of majority of the POSIX thread APIs (`pthread.h`), it enables verification of unmodified multithreaded programs. In particular, DiVinE explores all possible thread interleavings systematically at the level of individual bitcode instructions. This allows DiVinE, for example, to virtually prove an absence of deadlock or assertion violation in a given multlithreaded piece of code, which is impossible with standard testing techniques.

An invocation of DiVinE that performs assertion violation check for a multithreaded program, say `my_code.cpp`, is given below. First, C++ code is compiled into a LLVM bitcode file and then `divine verify` is used to execute a search for assertion violations.

```
$ divine compile --llvm [--cflags=" < flags > "] my_code.cpp
$ divine verify my_code.bc --property=assert [-d]
```

When no assertion violation is found, the same C++ code can be compiled into a native executable using the same tools and natively executed as follows.

```
$ clang [ < flags > ] -lpthread -o my_code.exe my_code.cpp
$ ./my_code.exe
```

This approach provides high assurance that the resulting binary meets the specification, since the bitcode can be verified post-optimisation. The only sources of infidelity are the native code generator (which is relatively simple compared to the optimiser) and the actual execution environment.

Without efficient state space reductions, the state space explosion stemming from the asynchronous concurrency of the very fine-grained LLVM bitcode would be prohibitive. Therefore, DiVinE comes with very efficient reduction algorithms (τ+reduction and heap symmetry reduction) [10] to facilitate verification. Efficiency of the reductions is indicated in Table 1.

The high level of assurance and a low entry barrier make this approach a very attractive combination. A set of examples (implemented in C and C++) which demonstrate the existing capabilities of the LLVM interpreter is distributed with DiVinE.

5 Timed Automata

Timed automata as used in Uppaal [7, 8] became a standard modelling formalism. The new release of DiVinE comes with the ability to perform LTL model checking and deadlock detection for real-time systems designed in Uppaal. On top of Uppaal Timed Automata Parser Library[2] (UTAP) and DBM Library[3], DiVinE implements an interpreter of timed automata, based on zone abstraction, see the scheme in Figure 1.

Both imported libraries and the new interpreter are built into the `divine` binary, allowing the tool to directly accept `.xml` files as produced by Uppaal IDE. For such the real-time systems, DiVinE is capable of performing time deadlock detection. Moreover, using the automata-based approach to LTL model checking, DiVinE allows verification of properties expressed as untimed LTL formulas over values of system data and clock variables. Since this approach does not distinguish zeno and non-zeno behaviours, some counterexamples may be spurious.

For untimed LTL model checking of real-time systems it suffices to provide the tool with an `.ltl` file of the same basename as the file describing the real-time system. If such a file is present when DiVinE is executed, it is automatically loaded and DiVinE offers to perform LTL model checking in addition to reachability analysis. Examples of real-time systems and corresponding LTL properties are part of the DiVinE distribution bundle.

6 Interface to External Interpreters

In version 3.0, DiVinE officially provides support for connecting third-party modelling formalisms. To this effect, DiVinE includes a model loader written to the Common Explicit-State Model Interface specification (CESMI). The CESMI specification defines a simple interface between the model-checking core and a loadable module representing the model. Generation of model states is driven by the needs of the model checking engine.

As a binary interface, CESMI requires a set of functions to be implemented in a form of dynamic (shared) library: this library is called a CESMI module. DiVinE's CESMI loader then connects the functions implemented in the module to the model checking engine: see also Figure 1. The two functions that must be implemented by all CESMI modules provide the initial states of the state space and generate immediate successors of any given state, respectively. A detailed

[2] `http://freecode.com/projects/libutap`

[3] `http://freecode.com/projects/libudbm`

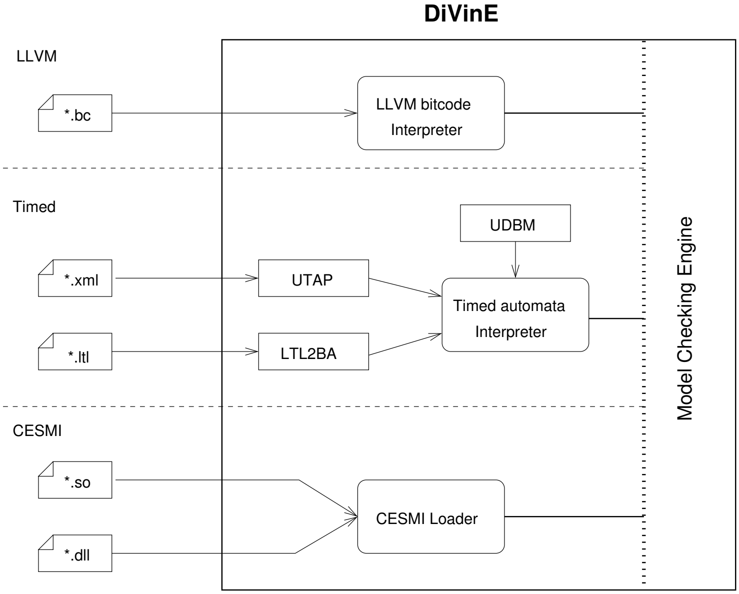

Fig. 1. Connecting DiVinE to new input languages

technical description of the interface is distributed with DiVinE. Note that the CESMI module takes different form depending on the target platform: ELF Shared Object files are supported on POSIX platforms, and Dynamically Linked Libraries (DLLs) on Win32 (Win64) platforms.

One of the advantages of using the CESMI interface in a third party project is that there is no need to implement an interpreter of the modelling language within DiVinE. In fact, new systems can be connected to DiVinE without changes to DiVinE itself, lowering the entry barrier for extending the tool.

A potential downside of the CESMI approach is that the CESMI module is responsible for presenting a Büchi automaton for the purposes of LTL model checking. While this requirement makes the CESMI specification more generic and flexible, it could present additional burden on the authors of CESMI modules. To mitigate this problem, DiVinE provides a small library of support code, automating both LTL conversion and construction of product automata. This functionality is available via the `divine compile --cesmi` sub-command and is documented in more detail in the tool manual.

The usefulness of the CESMI interface has been already demonstrated in several cases. First, we implemented a compiler of DVE (the native DiVinE modelling language) that builds CESMI modules and shows that a CESMI-based pre-compiled state generator is much faster than a run-time interpreter [5]. CESMI interface has also been successfully used in extending DiVinE to verify Murφ models [5]. More recently, the CESMI specification allowed us to build an interface between MATLAB Simulink and DiVinE, effectively creating a tool chain for verification of Simulink models [1].

7 Availability and Future Plans

DiVinE is freely available under BSD license. Stable releases as well as development snapshots and pre-releases are available for download at `divine.fi.muni.cz`.

Future development is expected to further improve scalability of the tool in parallel and distributed-memory settings. Moreover, we expect better state-space compression techniques and semi-symbolic model checking methods to again extend the applicability of DiVinE, to even larger and more complex systems. The set of C APIs offered by the LLVM interpreter will be expanded, extending the class of programs which can be verified without modification. An important future milestone is the addition of non-deterministic I/O and simulation of other system interactions.

References

1. Barnat, J., Beran, J., Brim, L., Kratochvíla, T., Ročkai, P.: Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 78–92. Springer, Heidelberg (2012)
2. Barnat, J., Brim, L., Ročkai, P.: DiVinE Multi-Core – A Parallel LTL Model-Checker. In: Cha, S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 234–239. Springer, Heidelberg (2008)
3. Barnat, J., Brim, L., Černá, I.: Cluster-based LTL model checking of large systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 259–279. Springer, Heidelberg (2006)
4. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: diVinE – A Tool for Distributed Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
5. Barnat, J., Brim, L., Ročkai, P.: DiVinE multi-core – A parallel LTL model-checker. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 234–239. Springer, Heidelberg (2008)
6. Barnat, J., Havlíček, J., Ročkai, P.: Distributed LTL Model Checking with Hash Compaction. In: Proceedings of PASM/PDMC 2012 (to appear 2013)
7. Behrmann, G., David, A., Larsen, K.G., Möller, O., Pettersson, P., Yi, W.: Uppaal - present and future. In: Proc. of 40th IEEE Conference on Decision and Control. IEEE Computer Society Press (2001)
8. Behrmann, G., Hune, T., Vaandrager, F.W.: Distributing Timed Model Checking - How the Search Order Matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 216–231. Springer, Heidelberg (2000)
9. Laarman, A., van de Pol, J., Weber, M.: Multi-Core LTSmin: Marrying Modularity and Scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
10. Rockai, P., Barnat, J., Brim, L.: Improved State Space Reduction for LTL Model Checking of C & C++ Programs. In: Submitted to The 4th NASA Formal Methods Symposium (2013)